第4章

CPUコアを検証時に活用して開発期間を短縮

JPEGエンコーダ回路の設計・検証

石田通彰

ここでは，FPGAを用いたJPEGエンコーダ回路の開発事例を取り上げる．画像処理のように扱うデータ量が多い回路の検証では，テストベンチが巨大になり，シミュレーションに膨大な時間が必要になる．また，カメラからの入力のように逐次変化するデータを扱う場合，実機による検証では処理するデータが毎回異なるため，処理の妥当性の判断がしにくくなる問題がある．そこで筆者は，汎用のFPGAボードとソフト・マクロのCPUを活用した設計・検証環境を作り上げた．開発期間の短縮に大きな効果があったという．（編集部）

今ではJPEG（Joint Photographic Experts Group）を応用した製品がいたるところにあふれています．ディジタル・カメラや携帯電話などの民生機器，複写機やプリンタなどのOA（office automation）機器，外観検査装置や分析装置などのFA（factory automation）機器，医療機器などです．まさに，「静止画あるところにJPEGあり」という感さえあります（**図1**）．

JPEGでは，静止画のデータ・サイズをおおむね1/10～1/100程度に圧縮できます．そこで，データの保存や通信の際にデータ圧縮（エンコード処理）を行います．データの復元（デコード処理）は，エンコード処理を行った機器が行う場合と，別の機器が行う場合があります．

JPEGのエンコード処理とデコード処理は，組み込みプロセッサのソフトウェアで実現する場合と，ハードウェアを用いる場合があります．システム構成や性能によりますが，VGA（640×480画素）サイズで15フレーム/s以上の処理を行おうとすると，現行ではハードウェア処理になるのではないでしょうか．近い将来，組み込みプロセッサの性能がさらに向上したり，あるいは高速アルゴリズムが発見されるなどして，VGAサイズで100フレーム/sでもソフトウェアで実現できる時代が来るかもしれません．しかしそのときには，より大きな画像への需要が増えているでしょう．大きな画像では，やはりハードウェア処理が求められます．

4

筆者は，ネットワーク対応の監視カメラ向けにJPEGエンコーダの開発を行いました．カメラ側でエンコード処理，パソコン側でデコード処理を行うシステム構成です．FPGA

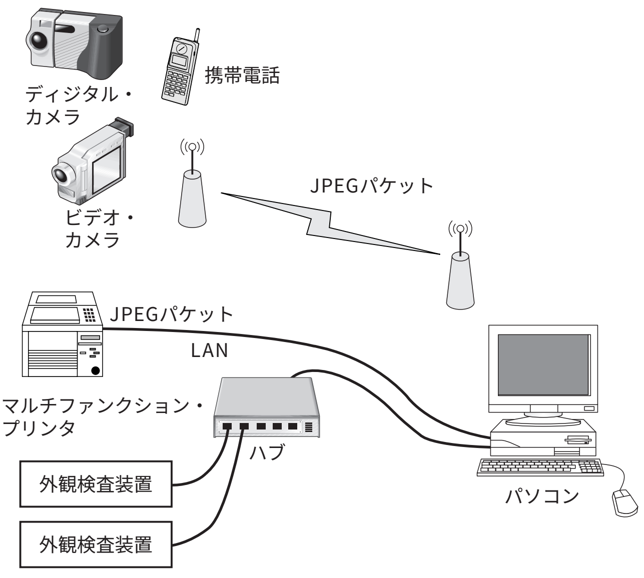


図1　JPEGを応用した製品
ディジタル・カメラや携帯電話などの民生機器，複写機やプリンタなどのOA機器，外観検査装置や分析装置などのFA機器，医療機器などで，JPEGがよく使われている．

KeyWord　JPEGエンコーダ，FPGA，CPU，離散コサイン変換，2次元ハフマン符号化，2ポート・レジスタ

(field programmable gate array)を使用してVGAサイズで30フレーム/s以上を実現することを目ざしました．

1．JPEGエンコーダの開発

論理回路の一般的な開発フローを**図2**に示します．

● アルゴリズムの調査と検証

最初にアルゴリズムを理解しなければなりません．JPEGエンコーダの開発におけるアルゴリズムの調査では，JPEGの規格書であるISO/IEC 10918-1やThomas G. Lane氏のソース・コード注1を使用しました．

JPEGの処理手順に関しては，教科書などで数多く取り上げられています．そこで，今回はエッセンスのみ簡単におさらいしたいと思います．

図3はJPEGの規格書に掲載されているエンコーダのブロック図です．

エンコード処理は，8×8画素のブロックを処理単位としています．周波数空間への変換には，離散コサイン変換(DCT：discrete cosine transform)を用います．

自然画像では画像の低周波成分が大きく，高周波成分は小さい特徴があります．また，高周波成分のノイズは人間の目にはわかりにくいものです．そこで，低周波成分に比べ，高周波成分は荒く量子化します．

データ量を小さくするために，DC成分については前のブロックとの差分値をハフマン符号化します．AC成分については，ランレングス符号化とハフマン符号化を組み合わせた2次元ハフマン符号化を使用します．また，ゼロラン長を大きくするために，ジグザグ・スキャンを使用します．

アルゴリズム検証用のプログラムはC言語で記述しました．これは，アルゴリズムの調査時に参考にしたコードがC言語だったことと，筆者の場合アルゴリズムの検証にはC言語を使用しているからです．

● ハードウェア化の検討

JPEGエンコーダの開発では，演算精度と画質注2の関係を求めるため，多くの画像を使って，それぞれ圧縮率を変えながら評価を繰り返す必要があります．そのため，論理シミュレーションの段階になってから評価を始めるのでは，

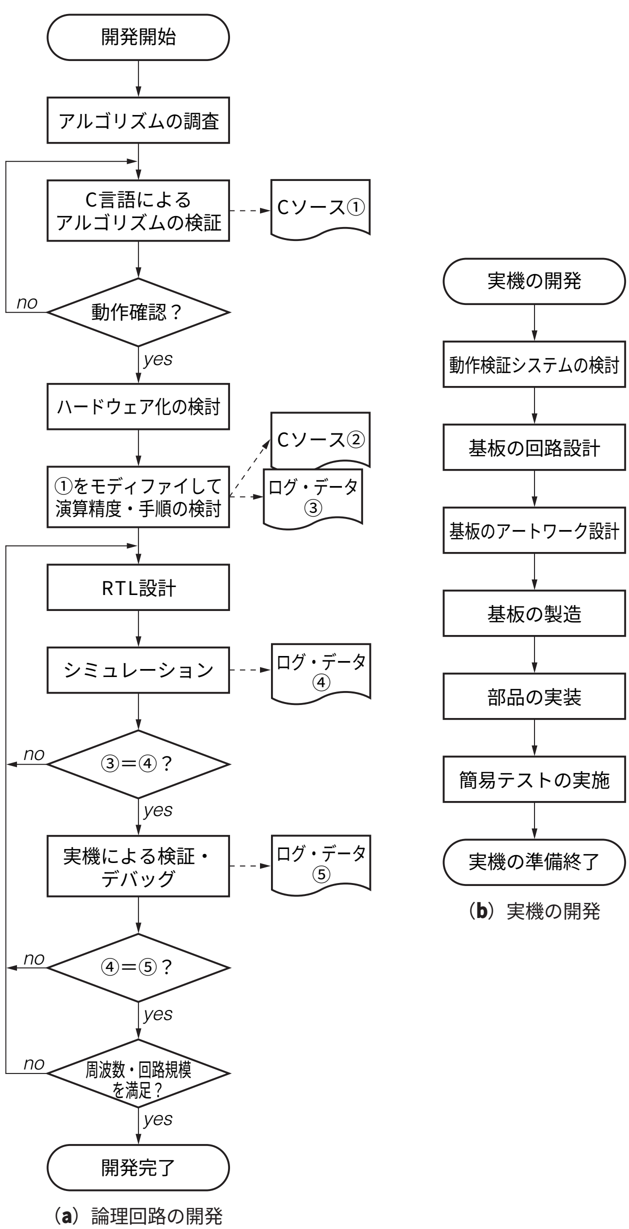


図2　論理回路の一般的な開発フロー

注1：Thomas G. Lane氏のソース・コードは，次のFTPサイトから最新版を入手できる．ftp://ftp.uu.net/graphics/jpeg/jpegsrc.v6b.tar.gz.
注2：今回はSN比を用いた簡易的な手法で画質を評価した．

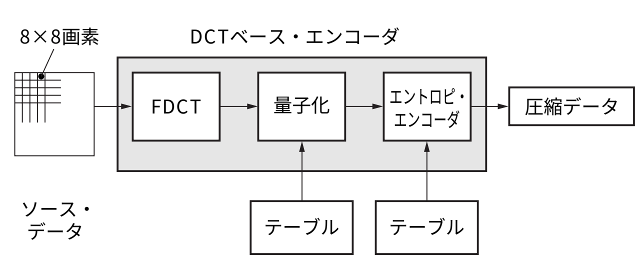


図3　JPEGエンコーダのブロック図

JPEGの規格書に掲載されているもの．FDCT＝forward DCT．

時間がいくらあっても足りません．

そこで，アルゴリズムの検証で用いたC言語プログラムをハードウェアの処理手順を意識して書き換え，演算精度と*SN*比との関係から演算精度を決定しました．これは通常の設計とは逆のプロセスになっています．アルゴリズム設計者がハードウェア構成の細部まで考えてプログラムを作成していない場合，このような逆プロセスが発生します．

具体的には，アルゴリズム検証では浮動小数点演算を用いていましたが，ハードウェア化の検討では固定小数点演算で丸め処理を行っています．また，実際のハードウェア処理と同じ手順とします．

例えば，DCT回路では，**C**をcos係数を要素とする8×8行列，$\mathbf{C}^t$をその転置行列，**Pixels**を8×8の画素値を要素とする行列として，

$$\mathbf{DCT} = \mathbf{C} * \mathbf{Pixels} * \mathbf{C}^t$$

の行列演算を行います．**図4**のように，FIFO（first-in first-outメモリ）からの信号到着順を考えると，最初に**Pixels**の1列目が同時に入力されるので，**C** * **Pixels** 演算が先になるように記述をします．今回の場合，アルゴリズム検証では，

$$\mathbf{DCT} = (\ \mathbf{C} * (\mathbf{Pixels} * \mathbf{C}^t))\ ;$$

と記述していたところを，ハードウェア検討時には，

$$\mathbf{DCT} = ((\mathbf{C} * \mathbf{Pixels}) * \mathbf{C}^t\ )\ ;$$

のように変更しています．

トップ・レベル・モジュールでCプログラムによる処理結果と論理シミュレーションの結果が完全に一致することを確認します．こうすることで，圧縮率と*SN*比の関係を求める際に，論理シミュレーションの代わりにCプログラムを使用できるようになります．数十～数百の画像に対するデータ収集時間を大幅に削減できます．また，検証・デバッグ時にCプログラムの結果を期待値データとしても使用できます（**図5**）．

今回は，監視カメラ用途を想定していたことから，画質を上げるよりも回路規模を抑えることを優先して演算精度を決めています．また，データ量がより小さくなるように2h2vサンプリング[注3]（YCbCr420）としています．

● シミュレーションとFPGA回路による検証

設計した回路は，まず，シミュレーションによる検証を行います．次に，FPGAに回路を実装して動作させることによる検証を行います．

FPGAによる検証では，検証用のボード（プリント基板）が必要になります．このボードは，最終システムにあわせて専用に設計するのが一般的です．この場合は，ボード開発期間やコストが問題になることがあります．また，ボードの検証について考えなければなりません．

そこで，市販のプロトタイプ開発システムを使用する方法も考えられます．しかし，多くの製品はかなり高価という問題があります．

汎用のFPGAボードを活用できれば，開発期間（納期）を短縮でき，開発コストも抑えられます．今回は，米国Altera社の「DSP開発キット　Stratix IIエディション」[注4]を使用しました（**写真1**）．

FPGAによる検証時には，シミュレーションで用いた画像データをパソコンから基板に転送してテストベクタとしています．また，内部信号や処理画像をパソコンに送り返

注3：T. G. Lane氏のドキュメントの表記．
注4：詳細はAltera社のWebサイト（http://www.altera.co.jp/products/devkits/altera/kit-dsp-2S60.html）を参照．

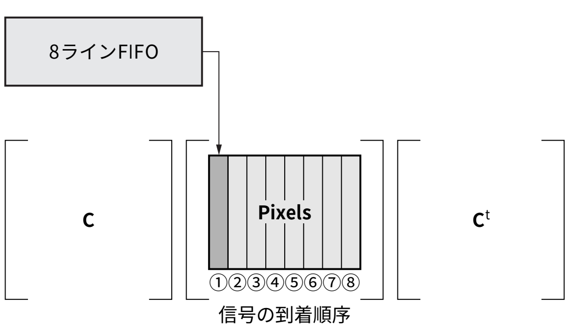


図4　FIFOからのデータの到着順
最初にPixelsの1列目が同時に入力される．

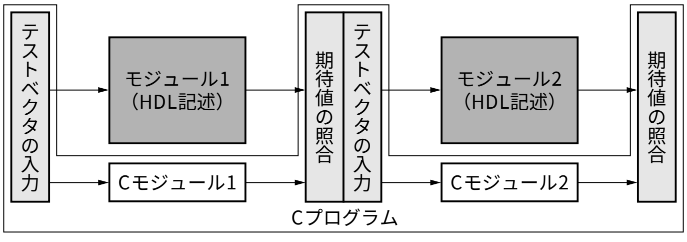


図5　Cプログラムを使用したモジュールのテスト
検証・デバッグ時にCプログラムの結果を期待値データとしても使用する．

4

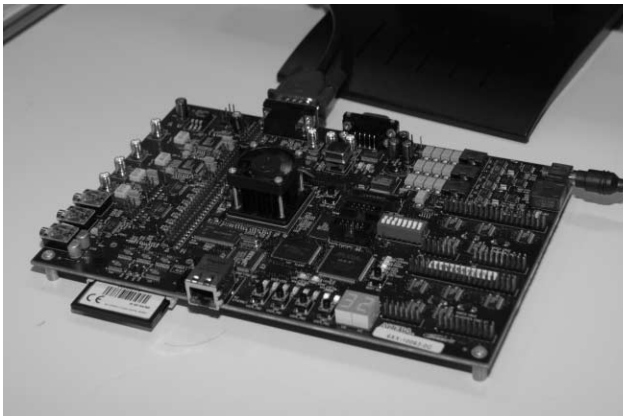


写真1　DSP開発キット　Stratix IIエディションのボード
FPGAとしてStratix IIファミリの「EP2S60」を搭載する．

すことで，デバッグ効率を上げました．

なお，Altera社のFPGAを使っていますが，テクノロジー依存性が極力なくなるように回路を設計しています．Altera社のマクロを使用したのはFIFOなどのメモリだけです．ほかはすべてプリミティブで回路を構築しました．とくに苦労したのはDCT回路です．積和演算の前段にブース・エンコーダ，中後段には3-2カウンタや4-2コンプレッサを用いた加算器をネットリストで直接記述しました(右掲のコラム「積和演算回路の設計」を参照)．

2. CPUコアを活用した検証システムの実現

一般的なLSI検証手法である論理シミュレーションには，非常に時間がかかります．現在の1万倍以上高速であってほしいと思うほどです．

● 画像処理ではシミュレーションによる検証が困難

JPEGエンコーダは，回路規模はそれほど大きくありません．しかし，1枚のVGA画像の処理に約300万クロックかかります．これをシミュレーションしようとしたら，かなりの時間が必要になります．100通りの画像についてシミュレーションをしようとすれば，100倍の時間が必要です．回路規模が100倍に，あるいはテストベクタ長が100倍になったらお手上げです．検証を高速に行うためのツールとして，シミュレーション・アクセラレータやエミュレータがありますが，非常に高価でだれもが使えるものではありません．

また，シミュレーションは動作を模擬しているにすぎず，LSIとしての完ぺきな動作を保証するものではありません．期待した性能で動作するかどうかもわかりません．

● 汎用FPGAボードを活用する

シミュレーションに要する時間と実デバイスによる動作の保証という二つの問題を解決するためには，身近な開発環境で，効率的に検証するためのシステムを構築するしかありません．すなわち，FPGAによる検証において，動作の妥当性を確実に判断できるしくみを作ればよいことになります．さらに，ソフトウェア・デバッガが持っているような機能をハードウェア開発でも活用できれば，さらに効率が上がるでしょう．

将来的には，実機やFPGAによる検証の比率を増やして，開発期間を短縮したいものです[注5]．そのためには，確実に，“楽しく”デバッグできる環境を構築することが必要です．

今回用いたFPGAによる検証システムの構成を**図6**に示します．指定の画像データを回路に入力したり，観測したい信号を選択するためにパソコンを用います．

● 画像の入力方法を考える

今回設計したJPEGエンコーダ回路は，CCDイメージ・センサやCMOSイメージ・センサから画像を入力することを想定した仕様になっています．

ところが，CCDイメージ・センサやCMOSイメージ・センサは，入手が難しい場合が少なくありません．筆者は以前，CMOSイメージ・センサのメーカにサンプル部品の問い合わせをしたことがあります．ところが携帯電話向けの量産計画がないと出荷できないと言われ，入手を断念しました．LSIの論理設計をしているからといって，量産まで手がけるとはかぎりません．

また，JPEGエンコーダの検証では，シミュレーション時に用いた画像データを実機でも使いたいという要求があります．実機の出力結果をシミュレーション結果と比較するためです．イメージ・センサなどの画像入力デバイスから，シミュレーション時に用いたのとまったく同じデータを受け取ることは不可能です．

そこで，今回は画像入力デバイスを使用せず，指定した

注5：実際の開発では，シミュレーションは不可欠である．しかし，実機やプロトタイプ・ボードをうまく使えば，シミュレーション回数を減らせると考えている．

テスト画像を検証対象の回路（DUT：device under test）に入力するためのしくみを用意しました（**図6**のDUT駆動回路）．

● 内部信号の観測方法を考える

回路の検証では，外部との入出力信号だけでなく，内部信号の状態も確認する必要があります．

ディジタル信号の観測では，ロジック・アナライザがよく使われます．FPGAの内部信号の観測では，FPGAの空きピンを使って信号を引き出すか，内部信号観測ツール（Altera社であればSignalTap Ⅱ）を利用します．

しかし，これらの方法では，観測する信号を変更するたびに，FPGAの回路を再コンパイルする必要があります．これはたいへんな手間であり，多くの時間を費やすことになります．

そこで，内部信号観測のための回路をあらかじめ埋め込んでおく手法をとりました．具体的には，観測したい信号をマルチプレクサで選択します．こうすることで，回路を再コンパイルすることなく，多くの信号を観測できるようになります．

COLUMN 積和演算回路の設計

3個以上のデータを加算する回路として，次の二つの構成法が有名です．

1) 3-2カウンタ，つまり，1ビット・フル・アダーで構成する（**表A-1**）
2) 4-2コンプレッサで構成する（**図A-1**）

今回は，両方のタイプでビット切り加算回路を構成して動作させましたが，システムの動作周波数がほかの要素で決まってしまい，差異を確認できませんでした．そこで，今回は初段の1次元DCTで使用する積和演算回路を3-2カウンタで，次段の積和演算回路を4-2コンプレッサで実装しました．一般的に，4-2コンプレッサが若干高速であると言われているようです．また，設計上も4-2コンプレッサがよりシンプルです．

本設計では加算回路にのみ3-2カウンタや4-2コンプレッサを使用していますが，乗算器の前段にブース・エンコーダではなく4-2コンプレッサを使用するほうが動作速度が上がるとの報告があります(A-4)，(A-5)．

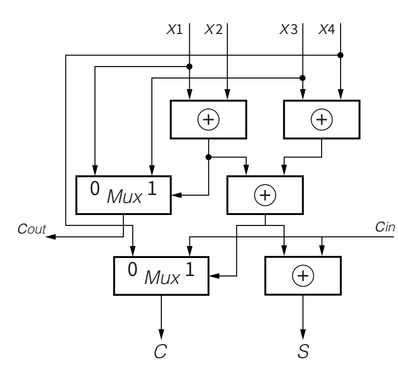
高速乗算回路の勉強にあたっては，オリジナル論文の一読をお勧めします(A-1)，(A-2)．なかなか時間がとれないかもしれませんが，多くの教科書や文献に引用されるような論文は一読しておくのが望ましいと思います．筆者も教科書の記述内容に理解や納得ができない場合にオリジナルの論文を読んで理解・納得できたことがたびたびあります．また教科書よりも論文のほうがていねいに説明されており，理解しやすいと思います．ただし，古い論文は入手困難な場合もあり，その場合は教科書(A-3)を使えばよいでしょう．最近では，Webサイトにもけっこう情報が出ています．

表A-1 3-2カウンタの段数

加算項数	3-2カウンタの段数
3	1
4	2
5 - 6	3
7 - 9	4
10 - 13	5
14 - 19	6
20 - 28	7
29 - 42	8
43 - 63	9
64 - 94	10
95 - 141	11
142 - 211	12
212 - 317	13
318 - 474	14
475 - 711	15
712 - 1066	16

参考・引用*文献

(A-1) A. D. Booth；“A signed binary multiplication technique”, Quart. J. Mech. Appl. Math., 4, Part2, pp.236-240, 1951.
(A-2) C. S. Wallace；“A suggestion for a fast multiplier”, IEEE Trans. on Computers, EC-13, pp.14-17, February 1964.
(A-3) Israel Koren；Computer Arithmetic Algorithms, A K Peters, 2001.
(A-4) P. Bonatto, V. G. Oklobdzija；“Evaluation of Booth's Algorithm for Implementation in Parallel Multipliers”, Twenty-Ninth Annual Asilomar Conference on signals, Systems and Computers, October 29 - November 1, 1995
(A-5) 外村元伸；回路設計者として身を立てるための“土台”を築く，Design Wave Magazine, p.37, 2006年5月号.



図A-1 4-2コンプレッサのブロック図

4

ハフマン符号化回路の出力(画像)は，FPGA内の32ビット×32Kワード構成(128Kバイト)のメモリを使用します(**図6**のメモリ1)．また，内部信号を観測するために，128ビット×4Kワード構成(64Kバイト)のメモリを使用します(**図6**のメモリ2)．

128ビット幅のメモリを使用しているので，信号は128本まで同時に観測できます．また，4Kワードの容量があるため1回の実行で4,096クロック分のログを記録できます．

1回のエンコード処理が終了後，出力画像(メモリ1)と観測信号(メモリ2)のデータを外部メモリ(**図6**の外部メモリ)にいったん転送し注6，その後パソコンに送り返します．

● FPGAボードをパソコンから制御

今回使用したFPGAボードには，外部インターフェースとしてRS-232-CとEthernetが搭載されています．今回は画像データを扱うため，高速なEthernetを使うことにし

注6：直接パソコンに転送することも可能．

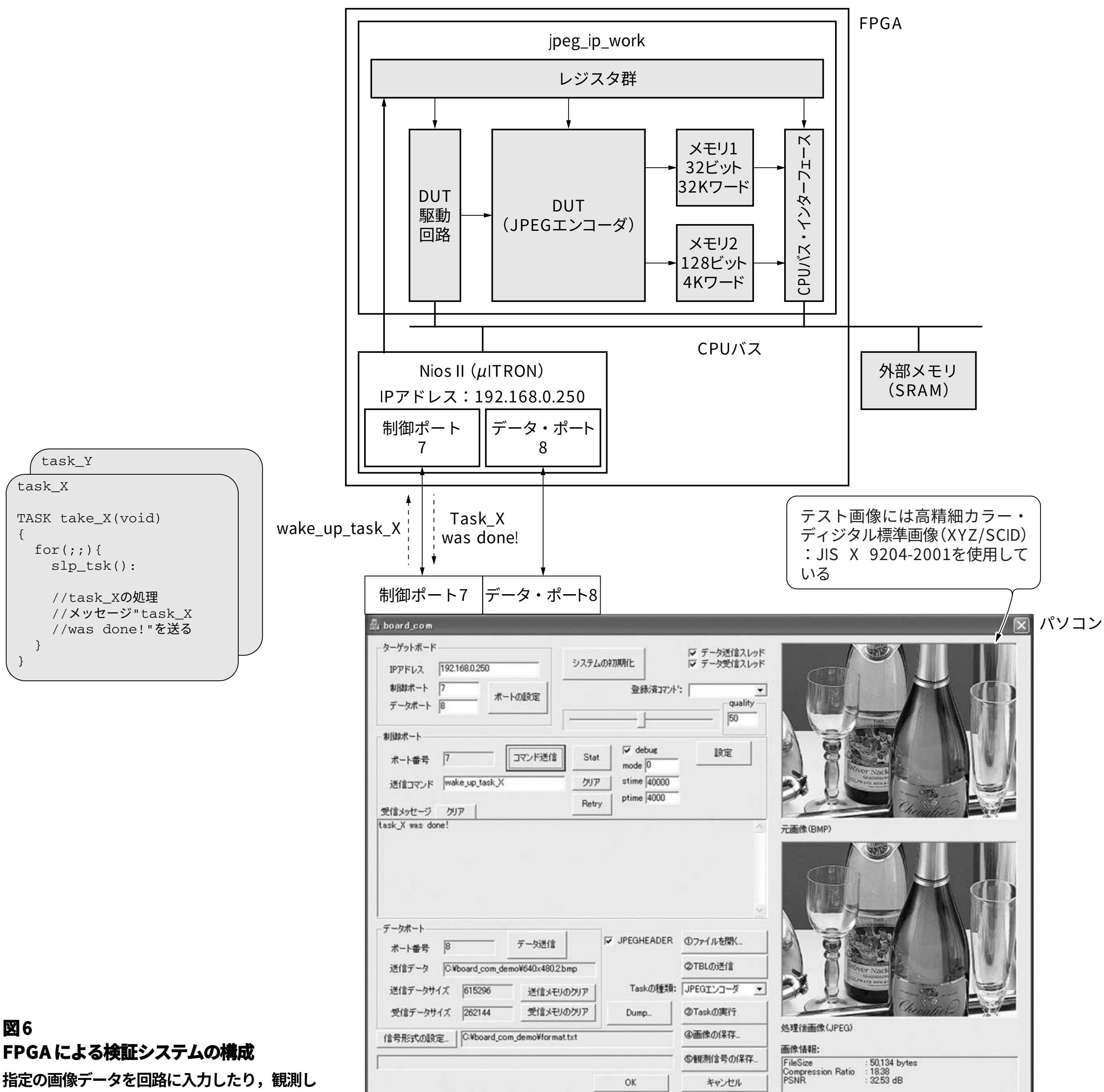


図6
FPGAによる検証システムの構成
指定の画像データを回路に入力したり，観測したい信号を選択するためにパソコンを用いる．

ます[注7].

実際の通信においては，パソコン側とFPGAボード側のソフトウェアが複雑にならないように，パケットを制御系とデータ系に分ける構成とし，それぞれ双方向通信ができるように設計しました（**図6**の制御ポートとデータ・ポート）.

制御ポートの送受信はデータ量が小さく，つねに1パケットの通信としているので，並列動作上も問題はありません．データ・ポートではパケット単位のデータ送受信が連続して発生しており，この間にもほかの処理を行えたほうがベターです.

FPGAボード側は，マルチタスク処理により並列動作できるようにします．パソコン側はデータ・ポートの送受信のみスレッド化することとしました[注8]．これは，筆者のようなせっかちな使用者は，データ送受信中に制御ポートを通して相手側と通信を行ったり，GUI（graphical user interface）を操作するからで，これらの処理がブロックされないようにするためです.

● CPUコアを使って信号の入出力を制御する

パソコン上のソフトウェアからFPGAを直接制御することはできません．そこで，FPGA内にソフト・マクロのCPU「Nios II」を実装し，Nios IIのソフトウェアとパソコン上のソフトウェアが協調動作するしくみを構築しました．こうすることで，パソコンから検証対象の回路を直接制御しているかのような使い勝手を実現できます.

Nios IIでは，マルチタスク処理のために，μITRON4.0仕様のリアルタイムOSであるミスポの「NORTi」を使用します．Nios II上のソフトウェアは，デバイス・ドライバ類のサイズが大きく，全体で約380Kバイトになりました．組み込みシステムとしては大きいと思います．しかし今回のような検証システムでは，ソフトウェアのサイズを抑えることが目的ではありません．また，ソフトウェアはFPGAボード上のSDRAMを使うことができます.

リアルタイムOSを使用しない場合，リソースの消費量は少なくてすみます．しかし設計が複雑になり，開発期間がかかります．仕様変更にもかなり手間がかかります．リアルタイムOSを使用すれば，追加仕様分を新たなタスクとして追加すればよいので，既存のソフトウェアを再構築する必要がありません.

パソコンとのインターフェースは，ハードウェアで構成することもできます．しかし検証システムとしての保守を考慮すれば，ソフトウェアに割り当てるようにくふうすべきです．ソフトウェアのコンパイル時間はFPGAのコンパイル時間に比べ，圧倒的に短くてすみます.

SRAMにデータを書き込んだり，読み出したりするだけなら，パソコン上のソフトウェア（プログラムA）とNios II上のソフトウェア（プログラムB）だけで十分です．しかし，これだけでは検証対象の回路（回路C）を制御できません．そのため，プログラムAが発行したコマンドや送信したデータをプログラムBが中継して回路Cに通知するしくみが必要になります．そこでプログラムAが送信したコマンドやテストベクタ以外の特殊なデータは，レジスタ群に保存しておく構成にします．このレジスタ群に，信号観測

注7：通信手段がRS-232-CであってもEthernetであっても，Windows側のソフトウェア開発工数はほとんど変わらない．FPGAに組み込むソフトウェアのほうが問題になる．Ethernetを使おうとすると，TCP/IPプロトコル・スタックが必要である．NORTiはプロトコル・スタックが付属している.

注8：静止画ではクリティカルな処理にはならないのでスレッド化しなかった．動画の場合はスレッド化が必要.

COLUMN　2ポート・レジスタの記述

シミュレーション時と異なる動作の原因となった2ポート・レジスタのVerilog HDL記述を**リストB-1**に示します.

signal_Aは，16ビット×8ワード構成のメモリを構成しています．1ショット・パルス（oneshot_pulse）が入ったときに，addr1番地にdata_Aを書き込みます．ところが，このメモリはつねに読み出し状態になっています．つまり，oneshot_pulseの入ったときには書き込みと読み出しの同時操作が発生しています.

筆者の場合，設計で多くの時間をロスするのはこうした初歩的なミスや勘違い，思い込みによるミスです．このようなミスを起こさない取り組み方を模索していますが，なかなか見つからないのが現状です.

リストB-1　問題になったVerilog HDLソース・コード

```
reg  [15:0] signal_A [0:7];
wire [15:0] data_A;
wire [15:0] data_B;

always @(posedge clk) begin
   if(oneshot_pulse) signal_A[addr1] <= data_A;
end
assign data_B = signal_A[addr2];
```

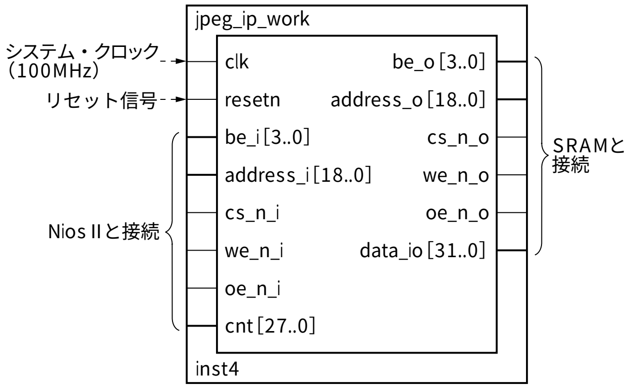


図7　Nios IIとJPEGエンコーダ回路の接続

レジスタ群との接続には28ビットの信号を用いている．上位8ビットを制御アドレスに，下位20ビットをデータに割り振った．

リスト1　インターフェース・レジスタの記述例

```
wire[7:0] logic_cnt = cnt[27:20];

always @(posedge clk) begin
  if (logic_cnt==8'h0) begin
      cnt_reg <= cnt[9:0];          //制御レジスタの設定
  end
  else if (logic_cnt==8'h20) begin
      ptime <= cnt[11:0];           //デバッグ期間の設定
  end
  else if (logic_cnt==8'hFF) begin //その他の設定
    {signal_1, signal_2,...,block1_start} <= cnt[19:0];
  end
end
```

```
KERNEL：00009e1b 0046 c6-c3-c8-bf 00-0-0 ffe4 ffe0
KERNEL：00009e1c fffc c2-b9-c4-bf 01-1-0 fffa ffe6
KERNEL：00009e1d ffff b3-af-ab-bc 05-2-0 ffff ffe8
KERNEL：00009e1e 0000 be-b8-b0-ab 06-3-0 fffa ffe8
KERNEL：00009e1f 0000 c1-c2-bd-b4 0e-4-0 0006 001f
KERNEL：00009e20 0000 c1-c1-c2-bb 0f-5-0 ffff 0038
KERNEL：00009e21 0000 c4-bf-c0-ba 1b-6-0 0000 003e
KERNEL：00009e22 0000 c7-c4-c3-bb 1c-7-0 ff9c 0030
 ⋮
中略
 ⋮
KERNEL：00009e53 0000 c7-b0-ae-ff 23-0-7 0014 00cd
KERNEL：00009e54 0000 c4-b9-cc-ff 24-1-7 fff4 00cc
KERNEL：00009e55 0000 c7-c2-c4-b4 30-2-7 fff9 00cf
KERNEL：00009e56 0000 c0-c2-c4-af 31-3-7 0000 00cd
KERNEL：00009e57 0000 bd-bf-bd-b4 39-4-7 fff9 ffed
KERNEL：00009e58 0000 c0-c2-b1-c3 3a-5-7 ffff fff0
KERNEL：00009e59 0000 bd-be-ac-e3 3e-6-7 0001 ffee
KERNEL：00009e5a 0000 c2-b0-a7-f5 3f-7-7 0249 fff5
```

時刻　DCT出力　入力信号　ジグザグ・スキャン　DCT内部ノード1　DCT内部ノード2

（**a**）シミュレーションのログ

```
00009e1b 0046 c6-c3-c8-bf 000-00-0 ffe4 ffe0
00009e1c fffc c2-b9-c4-bf 001-01-0 fffa ffe6
00009e1d ffff b3-af-ab-bc 005-02-0 ffff ffe8
00009e1e 0000 be-b8-b0-ab 006-03-0 fffa ffe8
00009e1f 0000 c1-c2-bd-b4 014-04-0 0006 001f
00009e20 0000 c1-c1-c2-bb 015-05-0 ffff 0038
00009e21 0000 c4-bf-c0-ba 027-06-0 0000 003e
00009e22 0000 c7-c4-c3-bb 028-07-0 ff9c 0030
 ⋮
中略
 ⋮
00009e53 0000 c7-b0-ae-ff 035-00-7 0014 00cd
00009e54 0000 c4-b9-cc-ff 036-01-7 fff4 00cc
00009e55 0000 c7-c2-c4-b4 048-02-7 fff9 00cf
00009e56 0000 c0-c2-c4-af 049-03-7 0000 00cd
00009e57 0000 bd-bf-bd-b4 057-04-7 fff9 ffed
00009e58 0000 c0-c2-b1-c3 058-05-7 ffff fff0
00009e59 0000 bd-be-ac-e3 062-06-7 0001 ffee
00009e5a 0000 c2-b0-a7-f5 063-07-7 0249 fff5
```

ジグザグ・スキャン

（**b**）実機テストのログ

```
MODE：0：11
          32,  ,H;
          16,  ,H;
           8,—,H;
           8,—,H;
           8,—,H;
           8,  ,H;
           8,—,D;
           4,—,D;
           4,  ,D;
          16,  ,H;
          16,  ,H;
END：0
```

10進数表示

（**c**）フォーマット変換設定ファイル

図8　検証結果

(a)と(b)ではジグザグ・スキャン部が異なった表示となっている，内容的には同じ．

期間を決めるレジスタ，あるいは，テスト対象をJPEGエンコーダ回路全体とするのか，特定の部分回路とするのかを選択するレジスタなどが含まれます．量子化テーブルもプログラムAから書き換えることができます．このときにレジスタ群を使用します．同じように，観測信号の選択状態もこのレジスタ群を使います．

task_Xは生成された後，起動させるとすぐにスリープ状態になります．プログラムAがプログラムBにtask_Xを起動するコマンド（wake_up_task_X）を送信した場合，プログラムBはtask_Xのwake_upコマンドを受信すると，task_Xの処理を1回だけ実行して再びスリープ状態に戻ります．

Nios IIと開発ターゲットを含むjpeg_ip_work回路との接続を**図7**に示します．レジスタ群との接続には28ビットの信号を用いています．上位8ビットを制御アドレスに，下位20ビットをデータに割り振りました（**リスト1**）．

● ホスト・パソコン用のソフトウェア

パソコン側のソフトウェアは，Windows上で，Visual C＋＋.NETを使って作成しました．

内部観測信号は，バイナリ・データになっています．バイナリ・エディタで見るのはわかりづらいので，信号フォーマット変換ファイルを参照して，シミュレーションと同

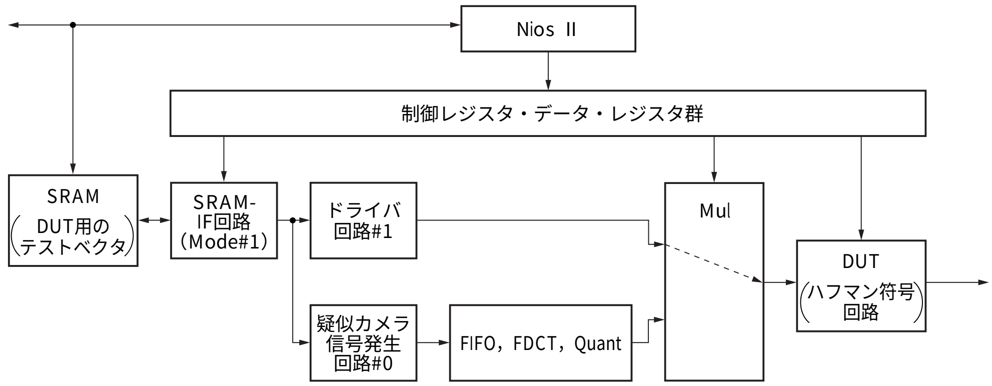


図9
サブブロックの検証
ハフマン符号化回路の検証では，SRAMにはハフマン符号化回路用の1MCU単位（16×16画素分）のテストベクタを入れる．

```
00000000─f4 000000000000000000000000
00000001─89 000000000000000000000000
00000002─6d 000000000000000000000000
00000003─cc 000000000000000000000000
00000004─e0 000000000000000000000000
00000005─88 000000000000000000000000
00000006─95 000000000000000000000000
00000007─a4 000000000000000000000000
00000008─6e 000000000000000000000000
00000009─85 000000000000000000000000
00000010─54 000000000000000000000000
00000011─67 000000000000000000000000
00000012─34 000000000000000000000000
00000013─d8 000000000000000000000000
00000014─b4 000000000000000000000000
00000015─eb 000000000000000000000000
：以下省略
```

内部カウンタ　内部ノード

（**a**）実機の出力

ADDRESS	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000280	F4	89	6D	CC	E0	88	95	A4	6E	85	54	67	34	D8	B4	EB
00000290	C8	86	2E	7C	9B	68	FB	34	F2	85	24	7D	07	35	60	5F
000002A0	5C	32	ED	FB	43	6D	C7	DD	5E	07	E9	55	2E	21	8E	E9
000002B0	4E	78	93	B3	1E	79	AC	D4	8B	B3	26	68	F4	E8	CE	5E
000002C0	EE	7B	86	1C	E2	DD	36	8C	FB	93	56	AC	AF	12	43	34
000002D0	36	96	B1	C2	7C	B2	E0	C9	99	0B	15	E8	39	AC	64	F3
000002E0	01	31	C8	BB	5C	75	F7	AB	9A	7C	82	DF	51	B6	90	9C
000002F0	2E	F0	A7	F1	E3	FA	D5	5C	56	19	36	A1	77	72	3F	7B
00000300	71	23	21	EC	A7	68	FC	85	79	5C	84	78	83	E3	4C	3B

（**b**）Cプログラムの出力

```
MODE：1：3
            24，－，D；
             8，  ，H；
            96，  ，H；
END：1
```

（**c**）フォーマット変換ファイル

図10
サブブロックの検証結果
FPGAの出力リストとCプログラムの出力を比較して同じ結果が得られた．

じ形式でファイル保存するように設計しました．

図8(a)が1ブロック分（64画素，64クロック）のシミュレーションの結果です．また，**図8(b)**がFPGAの内部信号観測結果です[注9]．**図8(c)**のフォーマット変換ファイルでは，実回路のログではジグザグ・スキャン部を10進数で表示するように設定しています．**図8(a)**と**図8(b)**ではジグザグ・スキャン部が異なった表示となっていますが，内容的には同じ結果です．

3．実際の検証

論理シミュレーション結果と実機による動作の結果は，一致していて当然と考えている人もいますが，なかなかそうとも言えません．一つには，ほんとうに内部回路が高い動作周波数で動作するのか，論理合成結果を信用できるのか，基板の電源余裕度やシグナル・インテグリティは十分かなどの問題があるためです．

今回も，最初にFPGAに回路を書き込んだ際には，正常に動作しませんでした．しかし，本稿で説明した検証システムを構築したことにより，1日で問題を特定し，解決することができました．

● シミュレーション時と異なる動作の原因を探る

最初は動作周波数を落として実験しましたが，バイナリ・レベルで完全に一致した不ぐあいが検出されました．そこで，論理合成に問題があると考えました．

まず不一致になる信号を後段のブロックから順番に，前段に向かって内部信号を観測していきました．その結果，あるHDL記述の解釈が，シミュレータと論理合成ツールで異なることがわかりました．ある回路ブロック内で2ポ

注9：画像システムの開発では，シミュレーションの結果をソフトウェアでチェックするので波形を見る機会はほとんどない．

4

ートのレジスタを記述している部分があり，Quartus IIでは期待とは別の回路を合成していることが判明しました(p.77のコラム「2ポート・レジスタの記述」を参照)．なお，EDAベンダからの情報では，別の論理合成ツールでは2ポート・レジスタを合成したとの報告を受けています．

そこで，2ポート・レジスタを回避する記述を行い，問題を解決できました．このようなデバッグを，再コンパイルを伴う手段で観測信号を追っていたのでは短時間で問題を解決できなかったものと思われます．

● サブブロックの検証

今回の検証システムでは，LSIを構成する一部のブロックだけを単独で検証することもできます．一例として，ハフマン符号化回路部分を単独で検証しました．

ハフマン符号化回路単体の検証には，**図9**のように，SRAMにはハフマン符号化回路用の1MCU単位(16×16画素分)のテストベクタを入れました．このテストベクタは前述したように，Cプログラム(**図2**の②)を使用して切り出したデータを使用しています．また，レジスタ群には，SRAMからハフマン符号化回路にデータが入るように設定します．それ以外の手順は先の例と同じで，ハフマン符号化回路単独の検証が行えました(**図10**)．

● さまざまな応用が考えられる

内部信号群の論理積や論理和でブレークポイントを設定する手法も実施しました．これ以外にもステップ実行やステップ実行の途中で内部信号の強制的な書き換えなど，いろいろ応用が考えられます．

このようなデバッグ手法を考えざるを得なかった理由は，開発全体に占めるシミュレーション時間やコンパイル時間の比率があまりにも大きく，なんとかならないものかとつねづね考えていたからです．

今回はシミュレーションが完了してから実機テストを実施しましたが，モジュール単位のシミュレーション段階で併用して使用すれば，全体での開発期間短縮につながりそうです．C言語などの高級言語によるアルゴリズム検証と並行して回路設計を進めているので，テストベクタの生成やシミュレーション結果との照合も高級言語を使って行えます．

シミュレータとプロトタイプ・デバッガがエレガントに融合したツールが将来登場してほしいものです．どのようなしくみが用意されるのか，そのときが楽しみです．

いしだ・みちあき
アオイ電子(株)